Windows User-Mode Drivers

Alex Ionescu
Chief Architect

Recon 2012

@aionescu
alex@crowdstrike.com

Bio

- Reverse engineered Windows kernel since 1999
  - Previously lead kernel developer for ReactOS Project
- Interned at Apple for a few years (Core Platform Team)
- Co-author of Windows Internals 5th and 6th Edition
  - Also instructor and contributor to Windows Internals seminar for David Solomon Expert seminars
- Founded Winsider Seminars & Solutions Inc., to provide services and Windows Internals training for enterprise/government
- Now Chief Architect at CrowdStrike

Introduction

Outline

- User-Mode Driver Framework (UMDF)
  - Architecture
  - UMDF 1.1
  - Requesting Direct Hardware Access for Fun and Profit
- RAM Attacks in VGA ROM BIOS
- HAL x86 eMulator (XM)
  - Initialization
  - Exported Interfaces
  - Access Rules
- The Attack
- Conclusion

Motivation

- XP
  - Published (along with many others) attacks on \Device\PhysicalMemory which allowed installation of call gates, system call hooking through KUSER_SHARED_DATA, and more…
    - Fixed in Server 2003
- Server 2003
  - Published (here at REcon) a bug in NTVDM VGA Frame Buffer mapping which allowed editing of arbitrary RAM (including kernel-mapped regions)
    - Fixed in Vista
- Vista/Windows 7
  - Published (at SyScan) issues in ACPI Override Tables and Watchdog Timer which allowed editing of arbitrary RAM (including kernel-mapped regions)
- Windows 8: UMDF 1.11 Allows access to RAM. One more attack?

Recommended Reading

- UMDF Guide (http://msdn.microsoft.com/en-us/library/windows/hardware/gg463294.aspx) -- Dev Center - Hardware > Docs > Drivers > Windows Driver Development > Windows Driver Frameworks > User-Mode Driver Framework

- IDE Port I/O (http://wiki.osdev.org/IDE)

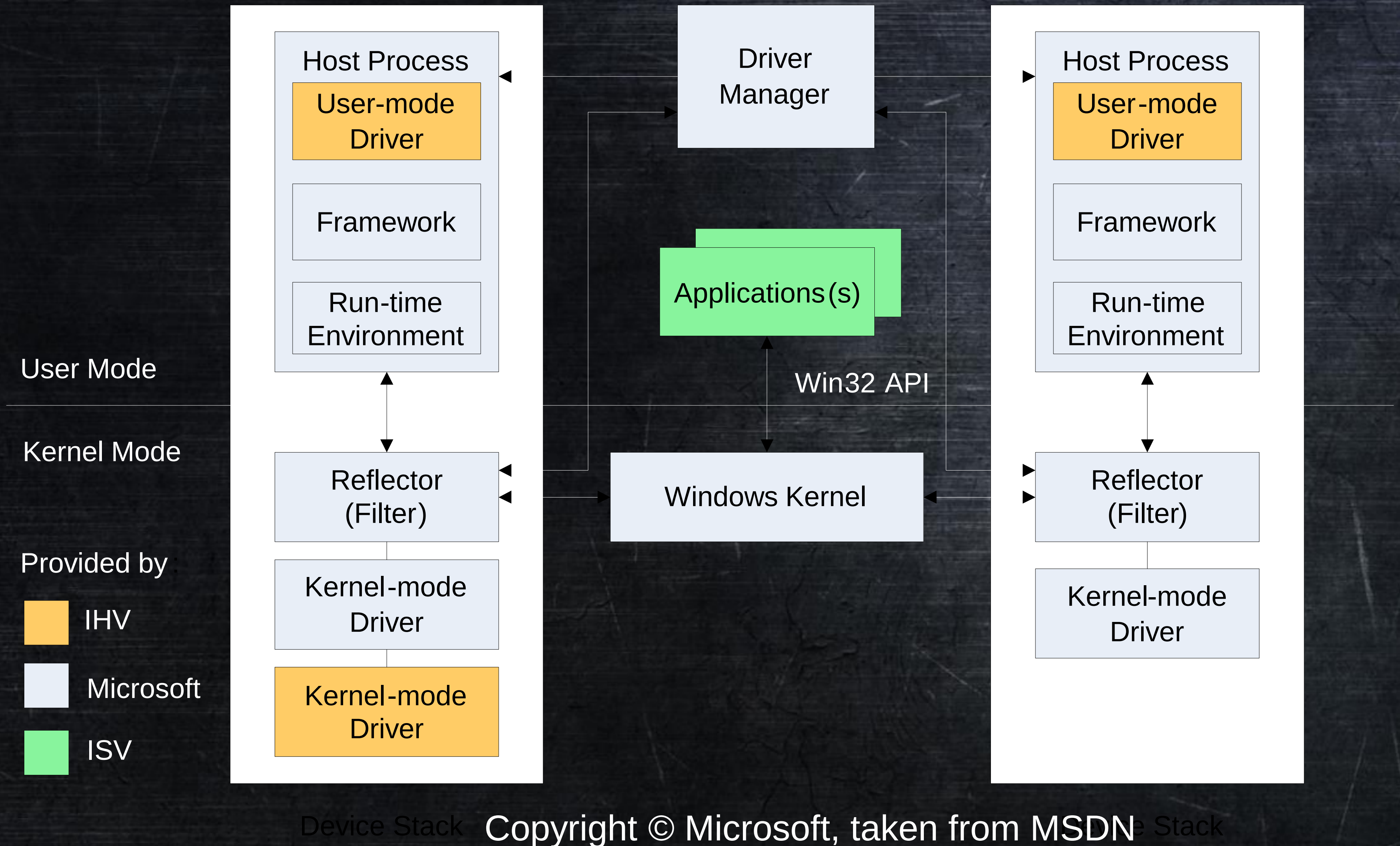
- Vmware High-Bandwidth Backdoor ROM Overwrite Privilege Elevation (http://packetstormsecurity.org/files/111404/VMware-High-Bandwidth-Backdoor-ROM-Overwrite-Privilege-Elevation.html)

User-Mode Driver Framework (UMDF)

UMDF Introduction

- Part of the Windows Driver Foundation (WDF)
  - Based on KMDF (Kernel-Mode Driver Framework)
- First released in Windows Vista, backported to Windows XP
  - KMDF backported all the way to Windows 2000!
- Designed for devices connected across a protocol bus (USB, 1394)
  - Portable storage devices, cell phones, MP3 players
  - Secondary displays over USB (such as Windows SideShow)
  - USB bulk devices
  - Touchscreens, etc…
- No interrupt support, and no access to hardware
  - Severely limits many other classes of devices
- Drivers are not subject to Code Integrity/Kernel Mode Code Signing

UMDF Architecture



Behavior of a UMDF Driver

- Runs inside a Driver Hosting Process
  - WUDFHost.exe
- Uses APIs from the UMDF Framework and Run-Time Environment
  - WUDFx.dll
  - WUDFPlatform.dll
- Managed by the UMDF Driver Manager running inside a Service
  - WUDFsvc.dll [Svchost.exe]
- Communicates with the kernel through…
  - ALPC
  - IOCTLs
- … to the UMDF Redirector
  - WDFRd.sys

UMDF 1.11

- Adds many new features to the framework, most importantly (for our purposes):
- The ability to handle interrupts in user-mode, both line-based (both level and edge triggered) and message-signaled
- The ability to map device registers in user-mode and access them directly
  - I/O ports are accessed through a system call
  - Memory-mapped I/O registers are accessed through a system call, but can be overridden to map the memory in user-mode directly!
- At SyScan, using similar access to MMIO registers by pretending to be a “Watchdog Timer”, was able to obtain Ring 0 persistence and code execution

Enabling Access to Hardware

- To get access to device registers, as well as to bypass the double-mapping and validation that is usually enabled by default, .INF file must contain:
  - [MyDevice_Install.NT.Wdf]
    UmdfDirectHarwareAccess=AllowDirectHardwareAccess
    UmdfRegisterAccessMode=RegisterAccessUsingUserModeMapping
- At this point, IWDFDevice3::MapIoSpace can be used
  - Check is done in user-mode, so malicious driver could bypass security by performing ALPC call directly to the WDF Reflector or by flipping internal bit on
- Wrote a small driver and attempted to replicate the SyScan ACPI attack, using the HAL Heap function table as a target
- However, was unable to map the required memory regions (was not sure why at the time), so spent time looking for other regions…

Memory Mapping Attempts

- Spent a few days attempting to map interesting regions of memory…
  - Tried almost all kernel/HAL/driver addresses -> FAIL
  - Tried low 1MB of memory -> FAIL
  - Tried framebuffer -> FAIL
  - Tried other device RAM/registers -> FAIL
- Finally decided to debug the failure
  - User-mode code in WUDFX.DLL (The Framework Library) ends up in CWdfCmResourceList::ValidateRegisterPhysicalAddressRange which checks if the device has any assigned registers
  - Patched code in memory to avoid checks, ALPC call got to the kernel!
- Kernel FAILed too ☹
  - WUDFRD.SYS RdCmResources::MapIoSpaceWorker has the same check

Driver Resource Allocation

- In the Windows I/O and PnP world, devices must request resources, and then go through a resource arbitration, translation, and assignment state machine
- Kernel ensures that all devices get the resources they requested, if possible
  - If not, kernel attempts to rebalance/arbitrate resources in order to make space
  - Most devices have “alternate” requirements as well, and some even have default states without any requirements
  - If all attempts fail, then the device does not receive resources and will fail to load
    - Device Manager shows exclamation mark
- Our driver is a software driver – no resources are assigned to it ☹

Bypassing or Forcing Resource Allocation

- As always, there is always some compatibility hack in Windows that gets you where you want to go…
- .INF Files can have a [LogConf] directive:
  - "A LogConfig directive references one or more INF-writer-defined sections, each of which specifies a logical configuration of hardware resources − the interrupt request lines, memory ranges, I/O ports, and DMA channels that can be used by the device. Each *log-config-section* specifies an alternative set of bus-relative hardware resources that can be used by the device"
- Specify MemConfig=YYYYYYYY@XXXXXXXX-ZZZZZZZZ
  - Where Y is length, X and Z are ranges to try finding the required length from
- Specify ConfigPriority=FORCECONFIG
  - Forces PnP manager to try assigning this configuration no matter what
- Sounds exactly like what we need – let's just hope the setting is honored even for UMDF drivers

Results

- Windows did honor the setting…
  - But refused to load the driver to a resource conflict
- Went back and tried different address ranges -> FAIL
- Finally tried MemConfig=1000@0-0xFFFFFFFF
  - The driver loaded!
- What resource did we get?
  - 0xC0000 (aka Video ROM BIOS)
  - Tried bumping up MemConfig to avoid this range -> FAIL
  - Out of the entire 4GB RAM address space, this was the only page Windows let the UMDF driver have
- “Now what the * am I supposed to do with this?”

RAM Attacks in VGA ROM BIOS

What can we do with RAM Access?

- Find out where kernel memory is mapped, and patch code
  - Subject to PatchGuard on 64-bit
- Find out where kernel objects are mapped (*NtQuerySystemInformation* or Win32k.sys) and patch those
  - One-bit in tagWND structure allows Ring 0 execution of arbitrary user-code on systems without SMEP enabled
- Etc…
- How do we translate to RAM?
  - In some cases, can leverage KSEG0 legacy mapping
    - i.e.: 0x80YYYYYY is 0xYYYYYY in RAM
  - Better approach: use undocumented SuperFetch API to do virtual->physical translation
    - Used by MemInfo and SysInternals RAMMap

Requesting RAM

- Create FX Device Object with IWDFDriver->*CreateDevice*
- Query IWDFDevice3 out of it with *QueryInterface* and the right IID
- Setup the register address in a PHYSICAL_ADDRESS structure
- Use *MapIoSpace* from IWDFDevice3 to obtain pseudo base address in user-mode.
- *GetHardwareRegisterMappedAddress* returns "real" address.

```
HRESULT
CMyDevice::Initialize(
    __in IWDFDriver           * FxDriver,
    __in IWDFDeviceInitialize * FxDeviceInit
    )
{
    IWDFDevice *fxDevice = NULL;
    IWDFDevice3 *fxDevice3 = NULL;
    HRESULT hr = S_OK;
    IUnknown *unknown = NULL;
    PHYSICAL_ADDRESS regsBasePA;
    PVOID m_RegBase = NULL;

    /* Get IUnknown */
    hr = this->QueryInterface(__uuidof(IUnknown), (void **)&unknown);
    if (FAILED(hr)) goto Exit;

    /* Create Framework Device */
    hr = FxDriver->CreateDevice(FxDeviceInit, unknown, &fxDevice);
    DriverSafeRelease(unknown);
    if (FAILED(hr)) goto Exit;
    fxDevice->Release();

    /* Get V3 Device Interface */
    hr = fxDevice->QueryInterface(__uuidof(IWDFDevice3), (void**) &fxDevice3);
    if (FAILED(hr)) goto Exit;
    m_FxDevice = fxDevice3;

    /* Map one page at 0xC0000 */
    regsBasePA.QuadPart = 0xC0000;
    hr = m_FxDevice->MapIoSpace(regsBasePA, 0x1000, MmNonCached, (void **)&m_RegBase);

    /* Release references and exit */
    fxDevice3->Release();
    DriverSafeRelease(fxDevice);
Exit:
    return hr;
}
```

VGA ROM BIOS

- Mapped at 0xC0000
  - JMP SHORT to INIT code
  - Magic sequence 0x55AA followed by vendor strings
- Registers Interrupt 10h in real-mode IVT
- Source code of vgabios used by most open source VM products:
  - http://cvs.savannah.gnu.org/viewvc/vgabios/vgabios.c?root=vgabios&view=markup

```
vgabios_init_func:

;; init vga card
  call init_vga_card

;; init basic bios vars
  call init_bios_area

#ifdef VBE
;; init vbe functions
  call vbe_init
#endif

;; set int10 vect
  SET_INT_VECTOR(0x10, #0xC000, #vgabios_int10_handler)

#ifdef CIRRUS
  call cirrus_init
#endif

;; display splash screen
  call _display_splash_screen

;; init video mode and clear the screen
  mov ax,#0x0003
  int #0x10
```

Attacking the VGA ROM BIOS

- Without access to IVT, how do we find where the INT10 handler is?
- One possibility:
  - Map the entire ROM
  - Scan for instruction sequence that is setting the IVT entry
    - VGA ROM BIOS is running on segment 0xC000:0000
    - But IVT is at 0x0000:0000
    - Which means that DS (Data Segment) must be switched by the code in order to access the IVT!
- Here’s the VGA ROM BIOS on my machine…

```
0:003> ur c0003 l1
000c0003 e9e600          jmp     00EC
0:003> ur c00ec
000c00ec e84634          call    3535
000c00ef e88f34          call    3581
000c00f2 1e              push    ds
000c00f3 31c0            xor     ax,ax
000c00f5 8ed8            mov     ds,ax
000c00f7 b81001          mov     ax,110h
000c00fa a34000          mov     word ptr ds:[00000040h],ax
000c00fd b800c0          mov     ax,0C000h
0:003> ur
000c0100 a34200          mov     word ptr ds:[00000042h],ax
000c0103 1f              pop     ds
000c0104 e8ce34          call    35D5
```

Triggering the Malicious Code

- VGA ROM BIOS executes in Windows when
  - Resolution is switched with a VGA video card driver that uses Video Port's INT10 interface
    - Usually only the Standard VGA Driver (Device Manager->Right click on Video Adapter->Disable)
  - Resolution is switched to full-screen mode in 16-bit application
    - But only allowed if Standard VGA Driver is running
  - The kernel crashes and causes a BSOD
    - But code execution now requires persistence – no way to "undo" the BSOD
  - Shutdown command is issued and the "It is now safe to Power Off your computer" is displayed (*DontPowerOffAfterShutdown* set in Registry)
  - Shutdown command is issued for a hibernate (to display hibernate UI)
- How is this code "executed"?

Real-Mode Code Execution on Windows

- Before Vista, Windows uses Virtual 8086 Mode to execute ROM code
  - A few bugs here over the years (Derek Soeder, Tavis Ormandy, myself)
  - *nt!Ke386CallBios* is used
  - “NTOSKRNL issues an INT 10h from a proper VDM with no interesting kernel code targets, but the VDM TIB is accessible to V86-mode code (at address 0x12000). The malicious INT 10h handler can modify the kernel stack pointer stored in 'CONTEXT.Esi', just as described in Tavis Ormandy's CVE-2010-0232 advisory […] in order to hijack execution after the cleanup code at NT!Ki386BiosCallReturnAddress completes.”
- Windows Vista and Windows 7 no longer use V8086 Mode unless HKLM\System\CurrentControlSet\Control\GraphicsDrivers\DisableEmulator key is set
  - *hal!x86CallBios* is used (TBD)
- Windows 8 always uses *hal!x86CallBios*

HAL x86 eMulator (XM)

XM Overview

- Originally implemented in MIPS, PPC, ALPHA HAL
  - Designed to support PC Video Card ROMs without vendor support
- Emulates x86 Real Mode
  - Instruction-level emulator
  - Support for 32-bit addressing and operands
  - Support for 486 instructions: BWAP, XADD, XMPXCHG
  - Support for 586 instruction: RDTSC
- Provides access to 16-bit address space through segmentation, with access to the low 1MB of memory (RAM)
  - Subject to restrictions (TBD)
- Provides access to PCI Bus and other hardware through I/O ports
  - Subject to emulation (TBD)

XM Initialization (*x86BiosInitializeBiosEx*)

- Initialized with 4 main addresses
  - “Transfer Memory”
    - Real-Mode<->Protected Mode Scratch Buffer
      - *x86BiosTransferMemory*
  - BIOS “IO Space Memory”
    - Base address of I/O addresses
      - *x86BiosIoSpace*
  - BIOS “IO Memory”
    - Base address of BIOS ROM
      - *x88BiosIoMemory*
  - BIOS “Frame Buffer”
    - Base address of VGA ROM
      - *x86BiosFrameBuffer*

XM Initialization (*HalInitializeBios*)

- During kernel initialization, *HalInitializeBios* calls *x86BiosInitializeBiosEx*
- Creates mapping for low 1MB
  - Removes any pages that are not marked as *LoaderFirmwarePermanent* or *LoaderSpecialMemory* by the boot loader
- Creates mapping for 0xA0000 to 0xC0000
- Creates mapping for 0x00000 to 0x00800
  - Copies data into low 1MB mapping, then frees this mapping

XM Memory Access Rules

- Implemented in x86BiosTranslateAddress
  - 0x90000 – 0x9FFFF and 0xC0000 – 0xFFFFF
    - Maps to BIOS EDA (Extended Data Area) and ROM, as well as VGA BIOS ROM
    - *x86BiosMemory* + 16-bit offset
  - 0xA0000
    - Maps to VGA Frame Buffer
    - *x86BiosFrameBuffer* + 16-bit offset
  - 0x00000 – 0x00800
    - Maps to real-mode IVT (Interrupt Vector Table)
    - *x86BiosLowMemory* + 16-bit offset
  - 0x10000 – 0x1FFFF and 0x30000 – 0x8FFFF
    - Returns 0
  - 0x20000 – 0x2FFFF
    - Maps to scratch buffer
    - *x86BiosTransferMemory* + 16-bit offset (limited to *x86BiosTransferLength)*

XM Port Access Rules

- Implemented in x86BiosRead/WriteIoSpace
  - 0xCF8 – 0xCFB
    - Maps to PCI Address Ports
    - Calls *x86BiosRead/WritePciAddressPort,* stored in *XmPCIConfigAddress*
  - 0xCFC – 0xCFF
    - Maps to PCI Data Ports
    - Calls *x86BiosRead/WritePciDataPort->XmGet/SetPciData->KdGet/SetPciDataByOffset*
  - 0x70 – 0x71
    - Maps to BIOS CMOS Ports
    - Calls *x86BiosRead/WriteCmosPort,* stored in *XmCmosAddress*
  - Remaining 64KB I/O Space
    - Direct Access to I/O Ports

Useful XM Tables

```
.text:800117D8 _XmOpcodeFunctionTable dd offset _XmAaaOp@4
.text:800117D8                                        ; DATA XREF: XmEmulateStream(x,x,x,x)+D0↓r
.text:800117D8                                        ; XmAaaOp(x)
.text:800117DC                 dd offset _XmAadOp@4    ; XmAadOp(x)
.text:800117E0                 dd offset _XmAamOp@4    ; XmAamOp(x)
.text:800117E4                 dd offset _XmAasOp@4    ; XmAasOp(x)
.text:800117E8                 dd offset _XmDaaOp@4    ; XmDaaOp(x)
.text:800117EC                 dd offset _XmDasOp@4    ; XmDasOp(x)
.text:800117F0                 dd offset _XmAddOp@4    ; XmAddOp(x)
.text:800117F4                 dd offset _XmOrOp@4     ; XmOrOp(x)
.text:800117F8                 dd offset _XmAdcOp@4    ; XmAdcOp(x)
.text:800117FC                 dd offset _XmSbbOp@4    ; XmSbbOp(x)
.text:80011800                 dd offset _XmAndOp@4    ; XmAndOp(x)
.text:80011804                 dd offset _XmCmpOp@4    ; XmCmpOp(x)
.text:80011808                 dd offset _XmXorOp@4    ; XmXorOp(x)
.text:8001180C                 dd offset _XmCmpOp@4    ; XmCmpOp(x)
.text:80011810                 dd offset _XmRolOp@4    ; XmRolOp(x)
.text:80011814                 dd offset _XmRorOp@4    ; XmRorOp(x)
.text:80011818                 dd offset _XmRclOp@4    ; XmRclOp(x)
.text:8001181C                 dd offset _XmRcrOp@4    ; XmRcrOp(x)
.text:80011820                 dd offset _XmShlOp@4    ; XmShlOp(x)
.text:80011824                 dd offset _XmShrOp@4    ; XmShrOp(x)
.text:80011828                 dd offset _XmIllOp@4    ; XmIllOp(x)
.text:8001182C                 dd offset _XmSarOp@4    ; XmSarOp(x)
.text:80011830                 dd offset _XmAndOp@4    ; XmAndOp(x)
.text:80011834                 dd offset _XmIllOp@4    ; XmIllOp(x)
.text:80011838                 dd offset _XmNotOp@4    ; XmNotOp(x)
.text:8001183C                 dd offset _XmNegOp@4    ; XmNegOp(x)
.text:80011840                 dd offset _XmMulOp@4    ; XmMulOp(x)
.text:80011844                 dd offset _XmImulxOp@4  ; XmImulxOp(x)
```

```
.text:80011948 _XmOperandDecodeTable dd offset _XmPushPopSegment@4
.text:80011948                                 ; DATA XREF: XmEmulateStream(x,x,x,x)+C1↓r
.text:80011948                                 ; XmPushPopSegment(x)
.text:8001194C                 dd offset _XmPushPopSegment@4 ; XmPushPopSegment(x)
.text:80011950                 dd offset _XmPushPopSegment@4 ; XmPushPopSegment(x)
.text:80011954                 dd offset _XmPushPopSegment@4 ; XmPushPopSegment(x)
.text:80011958                 dd offset _XmPushPopSegment@4 ; XmPushPopSegment(x)
.text:8001195C                 dd offset _XmPushPopSegment@4 ; XmPushPopSegment(x)
.text:80011960                 dd offset _XmLoadSegment@4 ; XmLoadSegment(x)
.text:80011964                 dd offset _XmLoadSegment@4 ; XmLoadSegment(x)
.text:80011968                 dd offset _XmLoadSegment@4 ; XmLoadSegment(x)
.text:8001196C                 dd offset _XmLoadSegment@4 ; XmLoadSegment(x)
.text:80011970                 dd offset _XmLoadSegment@4 ; XmLoadSegment(x)
.text:80011974                 dd offset _XmLoadSegment@4 ; XmLoadSegment(x)
.text:80011978                 dd offset _XmGroup1General@4 ; XmGroup1General(x)
.text:8001197C                 dd offset _XmGroup1Immediate@4 ; XmGroup1Immediate(x)
.text:80011980                 dd offset _XmGroup2By1@4 ; XmGroup2By1(x)
.text:80011984                 dd offset _XmGroup2ByCL@4 ; XmGroup2ByCL(x)
.text:80011988                 dd offset _XmGroup2ByByte@4 ; XmGroup2ByByte(x)
.text:8001198C                 dd offset _XmGroup3General@4 ; XmGroup3General(x)
.text:80011990                 dd offset _XmGroup45General@4 ; XmGroup45General(x)
.text:80011994                 dd offset _XmGroup45General@4 ; XmGroup45General(x)
.text:80011998                 dd offset _XmGroup7General@4 ; XmGroup7General(x)
.text:8001199C                 dd offset _XmGroup8BitOffset@4 ; XmGroup8BitOffset(x)
.text:800119A0                 dd offset _XmOpcodeRegister@4 ; XmOpcodeRegister(x)
.text:800119A4                 dd offset _XmLongJump@4 ; XmLongJump(x)
.text:800119A8                 dd offset _XmShortJump@4 ; XmShortJump(x)
.text:800119AC                 dd offset _XmSetccByte@4 ; XmSetccByte(x)
.text:800119B0                 dd offset _XmAccumImmediate@4 ; XmAccumImmediate(x)
.text:800119B4                 dd offset _XmAccumRegister@4 ; XmAccumRegister(x)
.text:800119B8                 dd offset _XmMoveGeneral@4 ; XmMoveGeneral(x)
.text:800119BC                 dd offset _XmMoveImmediate@4 ; XmMoveImmediate(x)
.text:800119C0                 dd offset _XmMoveRegImmediate@4 ; XmMoveRegImmediate(x)
.text:800119C4                 dd offset _XmSegmentOffset@4 ; XmSegmentOffset(x)
.text:800119C8                 dd offset _XmMoveSegment@4 ; XmMoveSegment(x)
.text:800119CC                 dd offset _XmMoveXxGeneral@4 ; XmMoveXxGeneral(x)
.text:800119D0                 dd offset _XmFlagsRegister@4 ; XmFlagsRegister(x)
.text:800119D4                 dd offset _XmPushImmediate@4 ; XmPushImmediate(x)
.text:800119D8                 dd offset _XmPopGeneral@4 ; XmPopGeneral(x)
.text:800119DC                 dd offset _XmImulImmediate@4 ; XmImulImmediate(x)
.text:800119E0                 dd offset _XmStringOperands@4 ; XmStringOperands(x)
.text:800119E4                 dd offset _XmEffectiveOffset@4 ; XmEffectiveOffset(x)
.text:800119E8                 dd offset _XmImmediateJump@4 ; XmImmediateJump(x)
.text:800119EC                 dd offset _XmImmediateEnter@4 ; XmImmediateEnter(x)
.text:800119F0                 dd offset _XmGeneralBitOffset@4 ; XmGeneralBitOffset(x)
.text:800119F4                 dd offset _XmShiftDouble@4 ; XmShiftDouble(x)
.text:800119F8                 dd offset _XmPortImmediate@4 ; XmPortImmediate(x)
.text:800119FC                 dd offset _XmPortDX@4   ; XmPortDX(x)
.text:80011A00                 dd offset _XmBitScanGeneral@4 ; XmBitScanGeneral(x)
.text:80011A04                 dd offset _XmByteImmediate@4 ; XmByteImmediate(x)
.text:80011A08                 dd offset _XmXlatOpcode@4 ; XmXlatOpcode(x)
.text:80011A0C                 dd offset _XmGeneralRegister@4 ; XmGeneralRegister(x)
.text:80011A10                 dd offset _XmNoOperands@4 ; XmNoOperands(x)
.text:80011A14                 dd offset _XmOpcodeEscape@4 ; XmOpcodeEscape(x)
.text:80011A18                 dd offset _XmPrefixOpcode@4 ; XmPrefixOpcode(x)
.text:80011A1C                 align 10h
```

Instruction Stream Emulation Main Loop

```
int __stdcall XmEmulateStream(USHORT Segment, unsigned __int16 Offset, _XM86_CONTEXT *Xm86Context)
{
  int status; // edx@1
  unsigned __int8 opCode; // al@3
  _OPCODE_CONTROL opcodeControl; // ax@3

  XmContext.Gpr[0].Exx = Xm86Context->Eax;
  XmContext.Gpr[1].Exx = Xm86Context->Ecx;
  XmContext.Gpr[2].Exx = Xm86Context->Edx;
  XmContext.Gpr[3].Exx = Xm86Context->Ebx;
  XmContext.Gpr[5].Exx = Xm86Context->Ebp;
  XmContext.Gpr[6].Exx = Xm86Context->Esi;
  XmContext.Gpr[7].Exx = Xm86Context->Edi;
  XmContext.SegmentRegister[3] = Xm86Context->SegDs;
  XmContext.SegmentRegister[0] = Xm86Context->SegEs;
  XmContext.SegmentRegister[1] = Segment;
  XmContext.anonymous_1.Eip = Offset;
  status = _setjmp3(&XmContext.JumpBuffer[4], 0);
  XmStatus = status;
  while ( !status )
  {
    XmContext.DataSegment = 3;
    LODWORD(XmContext.u.ControlInformation) = 0;
    HIDWORD(XmContext.u.ControlInformation) = 0;
    XmContext.OpcodeControlTable = (POPCODE_CONTROL)&XmOpcodeControlTable1;
    do
    {
      opCode = XmGetCodeByte(&XmContext);
      XmContext.CurrentOpcode = opCode;
      opcodeControl = XmContext.OpcodeControlTable[opCode];
      XmContext.OpcodeControl = opcodeControl;
      XmContext.FunctionIndex = opcodeControl.FunctionIndex;
    }
    while ( !XmOperandDecodeTable[XmContext.OpcodeControl.FormatType](&XmContext) );
    XmOpcodeFunctionTable[XmContext.FunctionIndex](&XmContext);
    status = XmStatus;
  }
  Xm86Context->Eax = XmContext.Gpr[0].Exx;
  Xm86Context->Ecx = XmContext.Gpr[1].Exx;
  Xm86Context->Edx = XmContext.Gpr[2].Exx;
  Xm86Context->Ebx = XmContext.Gpr[3].Exx;
  Xm86Context->Ebp = XmContext.Gpr[5].Exx;
  Xm86Context->Esi = XmContext.Gpr[6].Exx;
  Xm86Context->Edi = XmContext.Gpr[7].Exx;
  return status;
}
```

XM Interfaces

- Simple BIOS Call: *x86BiosCall*(interruptVector, biosContext)
  - Used by *VideoPortInt10* (see MSDN) and HAL for Blue Screen of Death

```
NTSTATUS __stdcall HalpBiosDisplayReset()
{
  _XM86_CONTEXT biosContext; // [sp+4h] [bp-20h]@1

  memset(&biosContext, 0, sizeof(biosContext));
  biosContext.Eax = 0x12u;
  return x86BiosCall(0x10u, &biosContext);
}
```

```
_XM86_CONTEXT   struc ; (sizeof=0x20, standard type)
_Eax            dd ?
_Ecx            dd ?
_Edx            dd ?
_Ebx            dd ?
_Ebp            dd ?
_Esi            dd ?
_Edi            dd ?
SegDs           dw ?
SegEs           dw ?
_XM86_CONTEXT   ends
```

- Complex BIOS Call: *x86BiosAllocateBuffer, x86BiosFreeBuffer, x86BiosReadMemory, x86BiosWriteMemory*
  - Used by VIDEO_PORT_INT10_INTERFACE
    - Call *VideoPortQueryServices*(VideoPortServicesInt10) to obtain
  - Implementations behind *Int10AllocateBuffer*, *Int10FreeBuffer*, *Int10ReadMemory*, *Int10WriteMemory*
  - See “Int10 Functions Implemented by the Video Port Driver” (MSDN)

XM Security

- Highly secure implementation from memory-access perspective
  - Multiple safeguards in place to ensure mapped memory is really valid BIOS/VGA ROM code and not kernel memory or undefined regions
  - However, BIOS memory is not *shadowed,* instead it is mapped with *MmMapIoSpace*
    - Writes will really write to BIOS memory
    - Changes to BIOS memory after HAL Initialization will be visible
    - Soft-reboot will maintain writes
  - Compare with Windows 7 NTVDM
    - BIOS memory is a read-write *copy* of real BIOS memory
- Wide-open to attacks from I/O-space access perspective
  - Access to PCI devices can allow PIO NIC access, for example
  - Also enables disk access through PIO IDE interface, for example

The Attack

HOWTO

- Write a UMDF 1.11 driver with direct hardware access enabled
  - However, only VGA ROM BIOS space seems obtainable
- Write attack/persistence code through mapped I/O addresses
  - However, code will be emulated by XM
  - Code will only execute if XDDM/Standard VGA Driver is used
    - Standard VGA Driver is WDDM driver on Windows 8
  - Must force resolution-change or blue screen of death to achieve code execution
    - Probably not going to work on EFI systems
- Must “escape” XM to affect actual machine
  - Only port I/O seems likely candidate
  - Requires legacy PIO IDE or NIC programming for persistence/backdooring
    - While making sure not to affect current use of hardware by the OS!

Sounds easy and reliable… right?

- If that sounded like it would
  - Take weeks of effort…
  - affect an increasingly smaller number of machines…
  - and require almost complete customization for a particular machine to work…
- That's because Microsoft did a good job
- Well played, well played…

Other Possibilities

- Writes to VGA ROM BIOS should persist across soft reboots
  - On some VMs, may persist on disk as well (due to bugs)
  - At reboot, code is executed natively, no XM present
    - Greater access to memory (can corrupt BIOS, ACPI tables)
    - Exclusive use of hardware, no worries about interfering with OS operation
  - Untested, but reported by other researchers to work
  - However, this means attack is only successful after machine reboot
    - Forcing reboot could raise user suspicion -- do the attack on Patch Tuesday? ☺
- Scratch buffer is initialized early on by *HalInitializeBios* and then used by VideoPort
  - *x86BiosAllocate/FreeBuffer* don't actually allocate/free anything!
    - Possible that some drivers depend on
      - VideoPort in Windows XP 64-bit had this issue, but the code is gone now
  - Idea is to corrupt the buffer from the attack code and attempt Ring 0 exec

Episode “8”: A New Hope?

```
8d5a7278 ffd6            call    esi {hal!x86BiosCall (817c65d2)}
8d5a727a 8ad8            mov     bl,al
8d5a727c 84db            test    bl,bl
8d5a727e 7466            je      BasicDisplay!BiosSetDisplayMode+0xae (8d5a72e6)
8d5a7280 66837dd84f      cmp     word ptr [ebp-28h],4Fh
8d5a7285 755f            jne     BasicDisplay!BiosSetDisplayMode+0xae (8d5a72e6)
8d5a7287 8d45d8          lea     eax,[ebp-28h]
8d5a728a 50              push    eax
8d5a728b 6a10            push    10h
8d5a728d c745d8034f0000  mov     dword ptr [ebp-28h],4F03h
```

Command - Kernel 'com:port=\\.\pipe\com_1,baud=115200,pipe,reconnect' - WinDbg:6.2.8400.0 AMD64

```
BasicDisplay!BiosSetDisplayMode+0x40:
8d5a7278 ffd6            call    esi
1: kd> k
ChildEBP RetAddr
a44af5f4 8d5a4b12 BasicDisplay!BiosSetDisplayMode+0x40
a44af610 8d5a48e5 BasicDisplay!BASIC_DISPLAY_DRIVER::SetSourceModeAndPath+0xb8
a44af65c 8d5a3970 BasicDisplay!BASIC_DISPLAY_DRIVER::CommitVidPn+0x279
a44af670 8d4740c5 BasicDisplay!BddDdiCommitVidPn+0x42
a44af690 8d473832 dxgkrnl!ADAPTER_DISPLAY::DdiCommitVidPn+0x44
a44af70c 8d4750f0 dxgkrnl!DmmCommitVidPn+0x25c
a44af8a0 8d476bd7 dxgkrnl!ADAPTER_DISPLAY::CommitVidPn+0x227
a44af8f8 8d48136c dxgkrnl!CommitVidPn+0x48
a44afaac 9bf3d146 dxgkrnl!DxgkCddEnable+0xae3
a44afb00 9bf3c653 cdd!CreateAndEnableDevice+0x18c
a44afc34 81464275 cdd!PresentWorkerThread+0x851
a44afc70 8132fdd1 nt!PspSystemThreadStartup+0x4a
00000000 00000000 nt!KiThreadStartup+0x19
```

DEMO

Conclusion

Key Takeaways

- UMDF 1.11 vastly increases the usability of the framework and the range of devices that can leverage it
  - Does so by adding user-mode interrupts and direct access to hardware
- Pros:
  - Less drivers in the kernel
  - Driver bugs become privilege escalation bugs, not Ring 0 bugs
    - Easier to mitigate against
    - Driver developers can choose to impersonate callers, and can even set maximum impersonation levels
  - Easier development, testing, debugging
    - Faster time to market for developers, faster access by users
- Cons
  - No Code Integrity (KMCS) validation of driver code ☹
  - Enables one kind of highly esoteric attack

Defense-in-depth Suggestions

- XM should make copies of BIOS/Firmware areas instead of mapping them
  - Would preserve compatibility (unless bizarre video card wants modification to survive across reboot?!)
  - Would prevent any kind of similar attack in the future from affecting VM/machine after reboot, or during resolution change
  - Will become nearly moot with Windows 8 and EFI
- XM could protect certain well-known I/O ranges or PCI devices from being accessed by VGA ROM BIOS
  - Potentially a lot of development effort to get right, probably not needed
- User-Mode Driver Framework should enforce KMCS in order to prevent unsigned drivers from loading!
- Why is PnP letting the UMDF driver map VGA ROM to begin with?

Trying out UMDF Development

- Download WDK 8.0 and Visual Studio 2012
- Obtain code sample (UMDF Driver Skeleton) http://code.msdn.microsoft.com/windowshardware/SKELETON-3a06c09e

QA

- Greetz/shouts to: Matthieu Suiche, Jason Geffner, Derek Soeder, Tarjei Mandt, Bruce Dang

CrowdStrike